Resourceful Lenses for Ordered Data

Aaron Bohannon
University of Pennsylvania

J. Nathan Foster
University of Pennsylvania

Benjamin C. Pierce
University of Pennsylvania

Alan Schmitt
INRIA Rhône-Alpes

Abstract

A lens is a bi-directional program. When read from left to right, it denotes an ordinary function. When read from right to left, it denotes an “update translator” that takes an input to this function together with an updated output and produces a new input reflecting the update.

Several variants of the idea of lenses have been explored. But none of them deal fully with ordered data. If an update involves a change in the order of a list (for example) in the output, the correspondence between the items in the output list and the items in the input list can be lost, leading to loss or corruption of data.

We propose a refined semantic space of resourceful lenses, enriching the well-behaved lenses of Foster, Greenwald, Moore, Pierce, and Schmitt (2007) with an equivalence relation embodying a notion of “reordering,” with respect to which the translation of updates must be invariant. We then present a regular-expression-like syntax for building resourceful lenses over string data. The key technical challenge is devising an intermediate semantic space that supports global matching of corresponding chunks of input and outputs structures while providing suitable denotations for a compositional language of lens combinators with local static checks guaranteeing that the semantic laws are satisfied. We give examples showing how our language can be used to build real-world string transformations.

1. Introduction

Most programs work in just one direction, from input to output. But sometimes, after calculating an output, we want to update this output and then compute in the other direction to find a corresponding updated input. Bi-directional languages address this problem by providing syntax in which every expression can be read from left to right as a map from inputs to outputs and from right to left as a map from updated outputs (together with an original input) to updated inputs.

Ordered data represents a significant challenge for bi-directional languages: even if the left-to-right calculation (which we call get) from input to output is order-preserving, when the updated input has been shuffled in some way, the backwards computation (put) needs to perform a more sophisticated, global matching of pieces of input and output data to accurately reflect the update on the input.

We make two main contributions. First, we refine the semantic space of lenses to a space in which invariants concerning the propagation of updated outputs to updated inputs under the shuffling of input data can be formulated. Then, starting from elementary string transductions specified by regular expressions, we develop a new bi-directional language for strings whose denotations lie in this space. Our design borrows the notion of provenance from databases. Programs identify chunks of the input and output structures that may be reordered and define how to compute a key for each chunk. This induces an association between inputs and outputs that guides the mapping from updated outputs back to updated inputs.

Our technical development is based on the presentation of lenses by Foster et al. (2007); we call these classic lenses to distinguish them from the other sorts of lenses we will define later. Formally, a classic lens mapping between a set of inputs C (also called “concrete structures”) and a set of outputs A (also called “abstract structures” or “views”) comprises three functions

\[
\begin{align*}
&l.\text{get} \in C \rightarrow A \\
&l.\text{put} \in A \rightarrow C \rightarrow C \\
&l.\text{create} \in A \rightarrow C
\end{align*}
\]

obeying the following laws for every \(c, c' \in C\) and \(a \in A\):

\[
\begin{align*}
l.\text{put} (l.\text{get} c) &= c \quad \text{(GETPUT)} \\
l.\text{get} (l.\text{put} a c) &= a \quad \text{(PUTGET)} \\
l.\text{get} (l.\text{create} a) &= a \quad \text{(CREATEGET)}
\end{align*}
\]

The mapping from inputs to views is given by the get component and may, in general, be a transformation that discards information. The put component takes as arguments an updated view and the original input, and weaves together the data from the view with information from the input discarded by the get component, to produce an updated input. The create component is like put, but is used to restore the information discarded by the get with defaults when the original input is not available. The laws ensure that the put and create components of each lens interoperate with the get component such that updates to views are propagated back to inputs in a reasonable way. The first law stipulates that the put function restores all of the information discarded by the get if its arguments are a view and an input that generates the very same view; the

---

1 Readers familiar with Foster et al.’s lenses may notice some small technical differences; these are discussed in Section 8.
second and third laws stipulate that the put and create functions do not discard any information contained in the view.

As a concrete example, consider the set of input structures given as comma-separated lists of ASCII text representing data about classical composers

"Jean Sibelius, 1865-1957, Finnish
Aaron Copland, 1910-1990, American
Benjamin Britten, 1913-1976, English"

and views containing just the name and nationality of each:

"Jean Sibelius, Finnish
Aaron Copland, American
Benjamin Britten, English"

The get component of a lens mapping between inputs and views is clear: for each line of the input it copies the name and nationality and deletes the years. There are several choices for the the put function, which combines the years in the input with the information in the view.

One possibility is to restore the year ranges by position—the name and nationality from each line in the view are combined with the years from the corresponding line in the input, using a default year range to handle cases where the view has more entries than the input. This policy reasonably reflects certain changes to the view back to the input; for example, if the update to the view replaces Britten’s nationality with “British” and adds an entry for Satie, then the put function combines

"Jean Sibelius, Finnish
Aaron Copland, American
Benjamin Britten, British
Erik Satie, French"

with

"Jean Sibelius, 1865-1957, Finnish
Aaron Copland, 1910-1990, American
Benjamin Britten, 1913-1976, English"

and yields:

"Jean Sibelius, 1865-1957, Finnish
Aaron Copland, 1910-1990, American
Benjamin Britten, 1913-1976, British
Erik Satie, ?????-????, French"

However, the behavior of this lens on other examples is not satisfactory. If the update to the view breaks the positional association between lines in the input and view, we do not get the output we expect—e.g., combining

"Jean Sibelius, Finnish
Benjamin Britten, British"

with the same input as above yields

"Jean Sibelius, 1865-1957, Finnish
Benjamin Britten, 1910-1990, British"

where the year data 1910-1990 has been restored from the entry for Copland!

What we would like is for the put function to align the entries in input and view by matching up lines whose name components are identical. On the inputs above, this put function would produce this result:

"Jean Sibelius, 1865-1957, Finnish
Benjamin Britten, 1913-1976, British"

Our goal is to build lenses with this sort of behavior.

To achieve this goal, two things are needed: semantics and syntax. First, we want a way of expressing, formally, the fact that the latter put function behaves well while the former does not. The refinement of lenses to resourceful lenses (R-lenses) gives us a vocabulary for requiring that the put function respects associations such as the one between entries having the same name in the input and view. Formally this is accomplished by extending classic lenses with an equivalence relation $\sim \subseteq C \times C$ and adding a law

$$\text{EQUIVPUT}$$

stating that the behavior of the put function on equivalent inputs is identical. In the above examples, if $\sim$ relates an input list to every list obtained by reordering its lines, then only the second put function obeys the new law.

Having identified this semantic space, we then instantiate it with a specific set of combinators for transforming strings. Their syntax is based on the regular expression presentation of finite state transducers, but also includes primitives for delimiting “chunks” of the input and view and for associating “keys” with each chunk. The put functions of combinator programs matches up chunks from the input with chunks from the view by comparing keys, but respecting the relative ordering of chunks with the same key. In the above examples, each line of input and view is a chunk; the first put function can be programmed by picking a constant as the key—i.e., every chunk has the same key and matches every other chunk, and the second by identifying the name data as the key. Every lens in this language has an associated equivalence relation $\sim$ that is calculated, compositionally, from its syntactic definition; the typing relation guarantees the classes lens laws as well as EQUIVPUT. (The combinators can also express, as a special case, a sub-language of classic lenses over strings in the style of finite state transducers, which is of interest in itself.)

We build up to our final combinator language in three steps. First we describe a set of basic combinators for transforming strings via regular operators— alternations, concatenation, and iteration—over primitives for copying and rewriting strings to a constant. These simple lenses do not know anything about chunks or keys, and so their behavior does not deal well with reordering. However, they are similar to the languages that follow, so we introduce them first to fix basic notation and to dispense with some well-formedness issues that are common to the languages that follow. Going back to our running example, the first put function that operates positionally is specified by the following lens program.

```lisp
let ALPHA = /[A-Za-z ]*/
let YEAR = /[0-9]{4}/
let YEARS = '/YEAR.\s\-\sYEAR/'
let c = ALPHA . "" .
   del YEARS . del\/, \ / .
ALPHA
let cs = "" | c . ("\n" . c)*
```

The first three lines define regular languages describing alphabetic and year data. The lens that processes a single line is c, and lists of entries are processed by cs. Within these lenses, the regular expressions and string constants denote copying lenses that operate as the identity on strings belonging to those sets in going from input to view and vice versa. The del lenses deletes the entire input in the get direction and restores it in the put direction (and create introduces a constant, "????-????"). As usual, alternation, concatenation, and iteration are written using |, . . and * respectively.

The second step defines an extension of these basic combinators— key lenses (K-lenses)—with components to calculate a key from inputs and views.

The third step is to show how to build R-lenses compositionally. We do this by defining a new family of lens-like structures—called skeleton lenses (S-lenses)—with components to identify chunks of inputs and the corresponding chunks of views. A top-level program—a resourceful lens—is then formed by combining a skeleton lens with a chunk lens. The skeleton lens matches up
chucks from input and view by key and then processes corresponding chunks using the chunk lens. Revisiting our example, the lens that matches composers by name is given by the following lens program:

```plaintext
let chunk =
  key ALPHA . ", " .
del YEARS . del \ / .
ALPHA

let cs_match = "" | [ _ ] . ("\n" . [ _ ])
```

The skeleton lens is given by `cs_match` and the key lens by `chunk`. It is almost identical to the simpler version above, except that the sub-lens that copies the name is wrapped in a `key` combinator, and the chunks are delimited by `[ ]`.

Our design is the first to apply the notion of provenance in a bi-directional language. In the database setting, provenance describes a relationship between the pieces of data in a view and the pieces of the source data that contribute to them. There are many different notions of provenance, depending on how “piece” and “contributes” are defined. For example, in the relational setting, “piece” is often taken at the granularity of tuples and “contributes” is calculated compositionally from the view definition (this is known as “why provenance”). In our setting, the identification of chunks, as specified by S-lenses, identifies the pieces of the concrete and abstract structures, and the nth chunk in the concrete structure is interpreted as contributing to the nth chunk in the abstract. This form of provenance, which we dub why-not provenance, controls the matching of chunks of concrete and abstract structure in the `put` function of S-lenses.

The rest of the paper is structured as follows. Section 2 fixes notion and formally defines classic lenses for strings based on regular operators. Section 3 describes our refinement of the semantic space classic lenses, yielding R-lenses; Section 4 likewise defines the semantic space of C-lenses and S-lenses structures and shows how these can be assembled to form R-lenses. Syntax and semantics for a specific selection of C-lenses and S-lenses combinators are given in Sections 5 and 6, respectively. Section 7 demonstrates the behavior of our prototype implementation on a realistic example—a lens mapping between XML and ASCII representations of business card data. Section 8 discusses related work. Section 9 sketches some extensions of the basic language.

### 2. String Lenses

Before we get to the details of our refined variant of lenses, let us complete the technical details of the language of classic lenses for strings that we used in the first example in the introduction. As that example showed, this language is not completely satisfactory in itself. Nevertheless, it is useful to define it in detail, both to establish notations that we will need later on and to give the reader a feel for how the slightly more complex languages of R-lenses and S-lenses will be defined below.

Let \( \Sigma \) be a fixed alphabet. A language is a subset of \( \Sigma^* \). Metavariables \( u, v, w \) range over strings in \( \Sigma^* \) and \( \epsilon \) denotes the empty string. The concatenation of two strings \( uv \) and \( v \) is written \( uvv \), and is lifted to languages \( L_1 \) and \( L_2 \) in the obvious way:

\[
L_1, L_2 = \{ uv \mid u \in L_1 \text{ and } v \in L_2 \}
\]

The notation \( L^* \) denotes the iteration of \( L \):

\[
L^* = \{ u_1 \cdots u_n \mid u_i \in L, 1 \leq i \leq n \}
\]

The well-definedness of the classic lenses presented later in this section depends on being able to split the elements of languages

-- because, in general, an input chunk matching some output chunk is not the one that it actually came from; rather, we attempt to identify an input chunk that a given output chunk "could have" come from.

In a unique way. The languages \( L_1 \) and \( L_2 \) can be unambiguously concatenated (written \( L_1 \cdot L_2 \)) if

\[
\forall u_1, v_1 \in L_1, \forall u_2, v_2 \in L_2, \quad u_1 \cdot u_2 = v_1 \cdot v_2 \implies u_1 = v_1 \land u_2 = v_2.
\]

A language \( L \) can be unambiguously iterated (written \( L^* \)) if

\[
\forall u_1, \ldots, u_m, v_1, \ldots, v_n \in L, \quad u_1 \cdot \cdots \cdot u_m = v_1 \cdot \cdots \cdot v_n \implies m = n \land \forall i \in \{1, \ldots, n \}, u_i = v_i.
\]

#### 2.1 Fact: It is decidable whether regular languages can be unambiguously concatenated and iterated.

Some of the classic string lens combinators will take regular expressions as arguments. These are written concretely according to the following grammar:

\[
\mathcal{R} ::= \ u \mid \mathcal{R} \cdot \mathcal{R} \mid \mathcal{R} | \mathcal{R} \mid \mathcal{R}^*
\]

where \( u \) is an arbitrary string. The notation \( [E] \) denotes the regular language described by the regular expression \( E \). Note that these regular expressions always denote non-empty languages. Given a regular expression \( E \), it is simple to identify an arbitrary member the language it denotes:

\[
\begin{align*}
\text{choose}(s) &= s \\
\text{choose}(E_1 \cdot E_2) &= \text{choose}(E_1) \cdot \text{choose}(E_2) \\
\text{choose}(E_1 | E_2) &= \text{choose}(E_1) \\
\text{choose}(E^*) &= \epsilon
\end{align*}
\]

Now we present five basic combinators for building lenses over regular languages. The simplest lens copies every string belonging to (the language denoted by) a given regular expression from the input to the view (and conversely in the opposite direction).

\[
\begin{align*}
\text{copy } E &\in \mathcal{R} \iff [E] \\
\text{(copy } E \text{).get } a &\equiv a \\
\text{(copy } E \text{).put } a &\equiv c \\
\text{(copy } E \text{).create } a &\equiv a
\end{align*}
\]

In programs, we write `copy E` as \( E \).

The second lens maps every string belonging to a given regular language to a string \( u \). In the `put` direction, it restores its input argument. To `create` an input from a view, we need a default \( v \), so that we can provide some member of concrete domain.

\[
\begin{align*}
\text{const } E \ u \ v &\in [E] \iff \{u\} \\
\text{(const } E \ u \ v\text{).get } a &\equiv u \\
\text{(const } E \ u \ v\text{).put } a &\equiv c \\
\text{(const } E \ u \ v\text{).create } a &\equiv v
\end{align*}
\]

Several useful derived forms can be expressed as syntactic sugar from `const`:

\[
\begin{align*}
\text{u} \leftrightarrow \text{v} &\in \{u\} \iff \{v\} \\
\text{u} \leftrightarrow \text{v} &= \text{const } u \ u \ v \\
\text{ins } u &\in \{\epsilon\} \iff \{u\} \\
\text{ins } u &= \epsilon \leftrightarrow \text{u} \\
\text{set } E \ u &\in [E] \iff u \\
\text{set } E \ u &= \text{const } E \ u \ \text{(choose}(E)) \\
\text{del } E &\in [E] \iff \{\epsilon\} \\
\text{del } E &= \text{set } E \ \epsilon
\end{align*}
\]

The next three lens combinators facilitate building bigger lenses from smaller ones and correspond to the basic regular operators:
updated view—otherwise, PUT
create
without the help of a concrete argument to guide it. To this end,
the lens has to push their contents back into the concrete domain
the extras. When there are more iterations in the abstract argument,
the same number of iterations of
the concrete argument, the result of the
is the root of the undesirable behavior we noticed in the exam-
ple
produced by the
of iterations of
A
produced by the
of iterations of
C
produce

The typing rule requires that the pair of input and pair of view do-
 mains can be unambiguously concatenated and is essential to en-
sure that the components of the lens are well-defined. For instance,
the lens

which is not typable using this rule has both

and

. We could attempt to implement a more complicated policy for choosing among the multiple parses
of strings, however, doing so would not give rise to lenses that satisfy
the lens laws in general. (The notation
also leaves implicit the assumption that
c and

must be chosen such that they are
members of

and

, respectively, and similarly for

and

. These conventions are used in the remainder of the paper.)

The combinator for building iterated lenses is similar.

There are two important things to note about the put component of
l*. First, it always calls the put component of l with iterations of A
and
C that have the same index, according to the unique parse. This
is the root of the undesirable behavior we noticed in the example
in the introduction. Second, it must handle cases where the number
of iterations of A found in the abstract argument is greater than
the number found in the concrete. As the number of iterations of
A produced by the get of l* is identical to the number of C in
the concrete argument, the result of the put function must have
the same number of iterations of C as there were iterations of A in
the updated view—otherwise, PUTGET will not hold. When there are
more iterations in the concrete argument, the lens simply ignores
the extras. When there are more iterations in the abstract argument,
the lens has to push their contents back into the abstract domain
without the help of a concrete argument to guide it. To this end,
every lens uses its create component.

The final combinator forms the alternation of two lenses:

3. Resourceful Lenses

We now define the semantic space of resourceful lenses and study
some of its abstract properties (i.e., properties that are independent
of our intended application domain of strings and the syntax of the
language of R-lenses for strings). Our main focus in this section is
on properties of the equivalence relation that can be used to
guarantee the association by key, instead of by position, of ordered
data in the put direction. Section 4 returns to the issue of defining
a syntax for (string) R-lenses.

An R-lens l from C to A is a tuple

satisfying the GETPUT, PUTGET, and CREATEGET laws from
Section 1 plus one additional law, for all

: C × C


The set of R-lenses from $C$ to $A$ is written $C \leftrightarrow A$.

As an example, let us take the R-lens from the end of the introduction, and assume for the moment that the concrete and abstract domains only include lists where no composer is mentioned twice—i.e., where the composer’s name functions as a unique identifier for each chunk. (This simplifying assumption will be lifted when we return to this example in Section 4.) A simple equivalence relation that would satisfy $\text{EQUIVPUT}$ would relate two lists of composers $c_1$ and $c_2$ whenever they differ only in their ordering. Since the $\text{put}$ direction of this lens returns a composer list whose order is the one from the abstract view, applying it to concrete elements that only differ by their order will yield identical results.

In the sequel depends on it.)

The syntax for string R-lenses that we will develop in the rest of the paper. (This discussion can be skipped on a first reading—nothing in the sequel depends on it.)

The only constraint on the $\sim_l$ relation is the $\text{EQUIVPUT}$ law; if we fix a triple of $\text{get}$, $\text{put}$, and $\text{create}$ functions satisfying $\text{GETPUT}$, $\text{PUTGET}$, and $\text{CREATEGET}$, there are, in general, many equivalence relations that satisfy $\text{EQUIVPUT}$. The finest relation (vacuously) satisfying $\text{EQUIVPUT}$ is the identity relation. The coarsest relation, written $\sim_{\text{UT}}$, is the equivalence induced by the $\text{EQUIVPUT}$ law (it is easy to check that this relation is an equivalence relation). Finer equivalences form a lattice between these two. In the following, we write $\text{Cl}(\sim_l)$ for the set of equivalence classes of $\sim_l$.

An intuition for which elements may be related by these equivalences can be extracted from the $\text{EQUIVPUT}$ law. By definition, if $c$ and $c'$ are equivalent, then, for every $a$, putting $a$ in $c$ and putting $a$ in $c'$ will yield the same result. Intuitively, this means that any difference between $c$ and $c'$ must be overridden by the put of $a$, and that any part of $c$ and $c'$ that is not overridden must be identical.

For instance, we gave earlier an example of a correct equivalence for the composers example, where we only allowed the order of composers to differ while having their birth date, death date, and nationality be the same in both lists. Still assuming each composer occurs at most once, one could leverage the fact that the lens overwrites the nationalities of composers to build a coarser equivalence where order and nationality would not matter: such an equivalence would relate concrete elements with the same sets of composers having the same birth and death dates in both equivalent elements, and one may show that it satisfies $\text{EQUIVPUT}$.

Going even further, we can actually calculate the coarsest possible equivalence for this lens, taking into account the creation of composers added in the abstract view yet not present in the concrete. These newly added composers have default birth and death dates, so it makes no difference if they are present in the original concrete argument with these defaults or absent: the result is the same. More precisely, two composer lists are related if the set of composers along with their (non default) birth and death dates are the same. Extra composers may occur in one list with the default birth and death dates as long as they occur with the same default dates or do not occur at all in the other list. In other words, within an equivalence class, the order of composers and their nationality may vary, and extra composers with exactly the default birth and death dates are allowed. I can be shown that this equivalence satisfies $\text{PUTPUT}$ and that, for every pair of concrete arguments $c$ and $c'$ that are not equivalent, there exists an abstract argument $a$ such that $\text{put } a \ c \neq \text{put } a \ c'$: this is the coarsest equivalence possible.

In some cases, that coarsest equivalence for a lens will actually be the universal relation. Such lenses are oblivious (in the sense of Foster et al.): the $\text{get}$ function is a bijection and the value of the concrete view does not matter in the $\text{put}$ direction. More generally, every concrete element $c$ may be characterized by the data preserved in the abstract domain (the abstract view $\text{get } c$) and the rest of the data shared by every other view of the equivalence class containing $c$. That is, given $C_i \in \text{Cl}(\sim_l)$ and an abstract view $a$, there is at most one view $c$ such that $c \in C_i$ and $\text{get } c = a$. Conversely, if two different concrete views map to the same $a$, then they must belong to different equivalence classes.

We now consider the relation between equivalence classes and series of successive updates. Some lenses have the property that each put completely overwrites the effects of earlier puts. In Foster et al. (2006), this situation is captured by an additional law stating that, for all $a, a' \in A$ and $c \in C$:

$$ l.\text{put } a \ l.\text{put } a' \ c = l.\text{put } a \ c \quad (\text{PUTPUT}) $$

Note that our new composers example does not satisfy $\text{PUTPUT}$: if one removes a composer with non-default dates and adds the same composer back immediately after, the birth and death dates will be the default ones instead of the original ones.

The $\text{PUTPUT}$ property may be characterized by a property of the coarsest equivalence of a lens $l$ from $C$ to $A$:

$$ l \text{ satisfies } \text{PUTPUT} \iff \forall C_i \in \text{Cl}(\sim_{\text{max}}) \ l.\text{get } C_i = A $$

There is indeed a tight relation between an equivalence class $C_i$ and its image under the $\text{get}$ function $A_i = \{l.\text{get } C_i \mid \text{for every } a \in A, \text{and } c \in C_i, l.\text{put } a \ c = c\_a \in C_i\}$, where $c\_a$ is the unique element of $C_i$ such that $l.\text{get } c\_a = a$. Hence, if $l.\text{get } C_i = A$, the result of any put remains in the equivalence class $C_i$ and is determined by the latest update, thus implying $\text{PUTPUT}$ for every $c \in C_i$. Conversely, if a lens obeys $\text{PUTPUT}$, then $c\_a \in C_i \in \text{Cl}(\sim_{\text{max}})$ implies $C_i = \{l.\text{put } a \ c\_a \mid a \in A\}$ and thus $l.\text{get } C_i = A$.

In our composers example, the only equivalence class of the coarsest equivalence whose image under $\text{get}$ is the entire abstract domain is the one where every composer has the default birth and death dates. For this special concrete structure, the effect of any put can be undone by another put. For all other equivalence classes, as soon as an edit removes a composer with non-default data, the updated concrete view switches equivalence class and no further edit will bring it back.

To close this discussion of generic properties of R-lenses, we propose a definition for the composition of R-lenses $l; k$ which is very close to the classic definition of Foster et al. (2007). Our contribution is a definition of $\sim_{\text{UT}(l; k)}$ that insures that $l; k$ also satisfies law $\text{EQUIVPUT}$.

$$ l \in C \leftrightarrow B \quad k \in B \leftrightarrow A \Rightarrow l; k \in C \leftrightarrow A $$

We have not yet found a need to include this operator in our implementation (we are not even completely sure it is pragmatically what one wants!), but one can imagine situations where defining R-lenses by composing others would be useful.

4. Building R-lenses

Our next task—which will take the rest of the paper—is to show how to build R-lenses (for strings) compositionally over the structure of the string type. This is a little tricky because we would sometimes like a chunk from one part of the concrete structure to be matched with a chunk from an entirely different part of the abstract structure, as we saw in the initial example. We address this
by defining a new form of lenses, S-lenses, whose behavior will conceptually have two phases when putting an abstract back into a concrete view. In the first phase, the concrete string is parsed into two pieces: the collection of chunks that may need to be globally reorganized, and a skeleton structure representing the parts of the string surrounding the chunks. The chunks that are collected are organized into a dictionary keyed by some information derived from the chunk itself. In the second phase, the put component of the S-lens will work structurally over the computed skeleton but will also be parameterized over a dictionary and return an updated dictionary, having removed any chunks for which it found a suitable match.

By contrast, all of the bidirectional transformations performed within a chunk are completely local. In fact, the plain string lenses presented earlier would almost suffice for writing lenses over chunks, except for one detail: the lens expression on the chunk must provide some specification of the information that identifies it. This information will be used by the S-lens’s put operation to determine how the chunk is matched, so it must be information that is present in both the concrete and abstract views of the chunk. The value used to identify a chunk will be called its key and will be used when building the dictionary structure to be passed to the S-lens. The lenses that operate on chunks will be called K-lenses and S-lenses will rely on a K-lens for processing the regions of the string marked as chunks. (For simplicity, we will assume that the same K-lens, called chunk, is used to process all chunks. We discuss how this assumption might be generalized to allow different K-lenses to be used on different chunks in Section 9.)

Since S-lenses depend upon K-lenses, we first present the definition of the latter. The definition of K-lenses is parameterized over a domain of keys $K$. For this paper, we will fix keys to be strings: $K = \Sigma^*$. A K-lens $l$ from $C$ to $A$, written $C \leftrightarrow A$ is a tuple of functions:

\begin{align*}
  l.\text{get} & \in C \rightarrow A \\
  l.\text{put} & \in A \rightarrow C \rightarrow C \\
  l.\text{create} & \in A \rightarrow C \\
  l.\text{key} & \in C \rightarrow K \\
  l.\text{akey} & \in A \rightarrow K
\end{align*}

satisfying the GETPUT, PUTGET, CREATEGET laws from above plus one additional law:

\begin{align*}
  l.\text{key} \ c = \text{Lkey} \ (l.\text{get} \ c) \quad (\text{GETKEY})
\end{align*}

We make the assumption that all chunks in the concrete domain of an S-lens have the same type and are processed by the same K-lens. Thus, we will parameterize our entire specification of S-lenses over a fixed regular language $R$, a fixed regular language $R_A$, and a fixed K-lens chunk $\in R \leftrightarrow R_A$. For simplicity, we will also assume the existence of a fixed string $r \in R$ that will be used as a default when one is needed.

Before we introduce the lens components of an S-lens, we must introduce several supporting structures and definitions. First, a dictionary over $R$ is a (total) function $K \rightarrow R$ list. The type of all dictionaries over $R$ is written $D$. Let $d \in D$ be a dictionary and $k \in K$ a key. We use the notation $\{k_1 \mapsto l_1, \ldots, k_n \mapsto l_n\}$ to denote the total function that maps $k_i$ to $l_i$, and every other key to the nil list. The concatenation of two lists $l_1$ and $l_2$ is written $l_1 \circ l_2$ and $[r]$ is the singleton list containing $r$. The concatenation of two dictionaries $d_1$ and $d_2$ is written $d_1 \circ d_2$ and defined as follows:

\begin{align*}
  (d_1 \circ d_2)(k) = d_1(k) \mapsto d_2(k)
\end{align*}

We note that this concatenation operation is associative. The update of a dictionary, written $d[k \leftarrow l]$ is defined as

\begin{align*}
  d[k \leftarrow l](k') = \begin{cases} 
    d(k') & \text{if } k \neq k' \\
    l & \text{if } k = k
  \end{cases}
\end{align*}

As mentioned earlier, an S-lens will need to build a representation of a string whose chunks have been removed. Concretely, these skeletons are strings over the alphabet $\Sigma \cup \{\square\}$, where we assume that $\square \notin \Sigma$. The boxes represent holes that used to contain elements of $R$. Conversely, we can talk about filling these with elements of $R$. We inductively define a relation $\rightarrow^*$ between strings over $\Sigma \cup \{\square\}$ and strings over $\Sigma$:

\begin{align*}
  \sigma \in \Sigma & \quad s \rightarrow s' \quad (\text{FILLSIGMA}) \\
  r \in R & \quad s \rightarrow r.s' \quad (\text{FILLBOX}) \\
  \square \rightarrow r & \quad s \rightarrow r.s' \quad (\text{FILLEPSILON})
\end{align*}

The completion of a language $S$ (written $\bar{S}$) is the set of all strings that can be build from a set of skeletons by filling in the holes:

\begin{align*}
  \bar{S} = \{ s' \mid s \in S \text{ and } s \rightarrow s' \}
\end{align*}

The completion operation has some important properties that are necessary to ensure that our typing rules in Section 6 are correct:

4.1 Lemma: If $\bar{S} = \emptyset$, then $S = \emptyset$.

4.2 Lemma: $S_1 \cup S_2 \subseteq (\bar{S}_1 \cup \bar{S}_2)$

4.3 Lemma: $S_1 \cap S_2 \subseteq (\bar{S}_1 \cap \bar{S}_2)$

4.4 Lemma: $\bar{S}_1 \subseteq (\bar{S}_2)$

4.5 Lemma: $\bar{S}^* \subseteq (\bar{S})^*$

Now we can finally define S-lenses. Let $S$ be a regular language over $\Sigma \cup \{\square\}$ and $A$ be a regular language over $\Sigma$. An S-lens $l$ from $S$ to $A$ is a tuple of functions

\begin{align*}
  l.\text{get} & \in \bar{S} \rightarrow A \\
  l.\text{put} & \in A \rightarrow \bar{S} \rightarrow D \rightarrow \bar{S} \times D \\
  l.\text{create} & \in A \rightarrow D \rightarrow \bar{S} \times D \\
  l.\text{parse} & \in \bar{S} \rightarrow D
\end{align*}

satisfying the following laws:

\begin{align*}
  s, d \mapsto l.\text{parse} \ c & \quad d' \in D \quad (\text{SGPUT}) \\
  l.\text{put} \ (l.\text{get} \ c) \ s \ (d \mapsto d') & = c, d' \quad (\text{SPUTGET}) \\
  l.\text{create} \ a \ d & \quad (\text{SCREATEGET})
\end{align*}

In order to get a better grasp on these structures, let’s consider the particular C-lens and S-lens associated with the example from the introduction. The fixed domain $R$ could be defined by the following regular expression:

```
/\^ALPHA/ . "/ "YEARS/ . " "\^ALPHA/
```

The fixed domain $R_A$:

```
/\^ALPHA/ . "/ "YEARS/ . " "\^ALPHA/
```

The fixed lens chunk would discard the years and would assert that the first \(^\wedge\text{ALPHA}\) region be a key. Now we can use the following regular expression to describe the set $S$:

```
" " \mapsto \square . ("\&\" " \mapsto \square) *
```

Then $\bar{S}$ would be the language described by the above expression where the boxes have been substituted for the expression describing
R. Now let’s consider the parse function we have in mind for this
S-lens. Calling parse on the concrete view

\[
\begin{align*}
\text{c} = & \\ & \{ \text{"Jean Sibelius", 1865–1957, Finnish} \\ & \text{\mid Aaron Copland, 1910–1990, American} \\ & \text{\mid Benjamin Britten, 1913–1976, English} \}
\end{align*}
\]

would result in the skeleton

\[
\begin{align*}
\text{s} = & \{ \text{\(_{\square}^1\) } \}
\end{align*}
\]

and the dictionary

\[
\begin{align*}
\{ & \text{"Jean Sibelius" \mapsto} \\ & \text{["Jean Sibelius", 1865–1957, Finnish"],} \\ & \text{\mid Aaron Copland" \mapsto} \\ & \text{["Aaron Copland", 1910–1990, American"],} \\ & \text{\mid Benjamin Britten", 1913–1976, English"} \}. 
\end{align*}
\]

Now assume we have the following updated abstract view:

\[
\begin{align*}
\text{a} = & \{ \text{"Benjamin Britten, British"} \\ & \text{\mid Aaron Copland, American} \}
\end{align*}
\]

The put function will traverse a\(_1\) and s simultaneously, until it reaches a \(_{\square}\) in s, at which point it will call the chunk.akey function on the corresponding substring of a\(_1\) for the purposes of retrieving data from the dictionary. For example:

\[
\begin{align*}
\text{chunk.akey } & \text{"Benjamin Britten, British"} \\ & \text{"Benjamin Britten"}
\end{align*}
\]

Having found an entry with this key in d, we want to fill in the box with the value computed by the chunk lens:

\[
\begin{align*}
\text{chunk.put} \\ & \text{"Benjamin Britten, British"} \\ & \text{"Benjamin Britten, 1913–1976, English"}
\end{align*}
\]

Our lenses are designed to work very intuitively when keys appear at most one time in the concrete view and at most one time in the abstract view. However, a type system based upon regular languages cannot ensure that this is the case. Instead of cluttering our type systems with such restrictions on the domains, it is possible to equip our lenses with a behavior that still obeys our lens laws in these questionable cases. This is facilitated by the fact that the range of the dictionary is a list of strings. For instance, consider the following concrete view:

\[
\begin{align*}
\text{c} = & \{ \text{"Benjamin Britten, 1913–1976, British"} \\ & \text{\mid Jean Sibelius, 1865–1957, Finnish} \\ & \text{\mid Benjamin Britten, 1913–1976, English"} \}
\end{align*}
\]

This will get parsed into the following dictionary:

\[
\begin{align*}
\{ & \text{"Jean Sibelius" \mapsto} \\ & \text{["Jean Sibelius", 1865–1957, Finnish"],} \\ & \text{\mid Benjamin Britten", 1913–1976, British"} \}
\end{align*}
\]

When there are multiple entries under a single key, the SGetPUT law will ensure that they are removed in a manner that respects their order.

**From S-lenses to R-lenses** Although the S-lens structure looks rather different from the classic lens structure or R-lens structure. Given an S-lens l, there is an easy way to build a corresponding R-lens which we denote l\(_{\hat{}}\):

\[
\begin{align*}
l \in C & \leftrightarrow A \\
\text{key } l & \in C \leftrightarrow A \\
\text{(key l).get } & = \text{ l.get} \\
\text{(key l).put } & = \text{ l.put} \\
\text{(key l).create } & = \text{ l.create} \\
\text{(key l).akey c } & = \text{ l.akey c} \\
\text{(key l).create c } & = \text{ l.create c}
\end{align*}
\]

In defining l\(_{\hat{}}\), we formally piece together the two phases described at the beginning of this section. Equality of parses induces an equivalence relation on S that we may use for the S-lens equivalence. This equivalence relates chunks such that keys occurring several times are related by position and these chunks need to be identical, but the order of chunks with distinct keys does not matter.

4.6 *Lemma:* l\(_{\hat{}}\in S \leftrightarrow A*.

**Proof:**

**GetPUT:** Let c \in S and s, d = l.parse c. We calculate as follows:

\[
\begin{align*}
l \text{.put (l.get c) } & = \pi_1(l.l.put (l.l.get c) s d) \\
& = c & \text{By SGGetPUT for l.}
\end{align*}
\]

**PutGET:** Let a \in A and c \in S and s, d = l.parse c. We calculate as follows:

\[
\begin{align*}
l \text{.get (l.put a c) } & = \pi_1(l.l.get (l.l.put a s d)) \\
& = a & \text{By SPGetPUT for l.}
\end{align*}
\]

**CREATEGet:** Let a \in A. Then

\[
\begin{align*}
l \text{.get (l.create a) } & = l.l.get (l.l.create a) \\
& = a & \text{By SCREATEGet for l.}
\end{align*}
\]

**EQUIVPUT:** Let a \in A and c, c' \in S with c \sim_i c'. Then there exists s \in S and d \in D with s, d = l.parse c = l.parse c'. We calculate as follows:

\[
\begin{align*}
l \text{.put a c } & = \pi_1(l.l.put a s d) \\
& = l.l.put a c' & \square
\end{align*}
\]

5. Syntax for K-lenses

A syntax for K-lenses can be designed to look very close to that for plain string lenses. However, we will need to introduce one special combinator for declaring a region of a chunk to be a key.

\[
\begin{align*}
l \in C & \leftrightarrow A \\
\text{key } l & \in C \leftrightarrow A \\
\text{(key l).get } & = \text{ l.get} \\
\text{(key l).put } & = \text{ l.put} \\
\text{(key l).create } & = \text{ l.create} \\
\text{(key l).akey c } & = \text{ l.akey c} \\
\text{(key l).create c } & = \text{ l.create c}
\end{align*}
\]

This construction takes on exactly the behavior of its lens argument, except that it sets of the key of the region to be the value of the abstract structure. It satisfies GETKEY trivially.

The other K-lens combinators are simply enriched versions of the combinators introduced in Section 2 for classic string lenses. We assume they have exactly the same typing rules (replacing the
classic lens arrow with the K-lens arrow). We need only to define the ckey and akey functions for each combinator, and demonstrate that the typing rules ensure the GETKEY property.

\[
\begin{align*}
\text{(copy R).ckey } c &= \epsilon \\
\text{(copy R).akey } a &= \epsilon \\
\text{(const R s d).ckey } c &= \epsilon \\
\text{(const R s d).akey } a &= \epsilon \\
(l_1 \cdot l_2).ckey (c_1 \cdot c_2) &= (l_1.ckey c_1) \cdot (l_2.ckey c_2) \\
(l_1 \cdot l_2).akey (a_1 \cdot a_2) &= (l_1.akey a_1) \cdot (l_2.akey a_2) \\
\end{align*}
\]

These combinators do nothing interesting with the key functions. It is straightforward to show that they satisfy the GETKEY law.

The union combinator will need to do something slightly more interesting with the key functions. Given lenses \( l_1, l_2 \in C_1 \xrightleftharpoons{} A_1 \) and \( l_2 \in C_2 \xrightleftharpoons{} A_2 \), we define:

\[
\begin{align*}
(l_1 \mid l_2).ckey &= \begin{cases} 
    l_1.ckey c & \text{if } (l_1 \mid l_2).get c \in A_1 \setminus A_2 \\
    l_2.ckey c & \text{if } (l_1 \mid l_2).get c \in A_2 \setminus A_1 \\
    \epsilon & \text{if } (l_1 \mid l_2).get c \in A_1 \cap A_2
\end{cases} \\
(l_1 \mid l_2).akey &= \begin{cases} 
    l_1.akey a & \text{if } a \in A_1 \setminus A_2 \\
    l_2.akey a & \text{if } a \in A_2 \setminus A_1 \\
    \epsilon & \text{if } a \in A_1 \cap A_2
\end{cases}
\end{align*}
\]

In the case of a union, we do not require that \( A_1 \) and \( A_2 \) are disjoint, and if we apply \( l_1.akey \) and \( l_2.akey \) to an element of \( A_1 \cap A_2 \), it is possible that we will get different answers. We cannot arbitrarily choose one and expect to satisfy GETKEY. So we make akey the constant function (returning the empty string) on these inputs, and make ckey the same on corresponding elements of the concrete domain, so that GETKEY is satisfied.

6. Syntax for S-lenses

While the semantics of S-lenses is somewhat more complicated than for plain string lenses, the syntax for building them is not. We again have copy and const combinators take the same arguments and perform the same function. We will use the same syntactic sugar for these as we do for plain string lenses.

\[
\begin{align*}
E \in R & \quad \text{copy } E \in \parallel E \xrightleftharpoons{} E \\
(copy E).get c &= c \\
(copy E).put a s d &= a, d \\
(copy E).create a d &= a, d \\
(copy E).parse c &= \{ \}
\end{align*}
\]

\[
\begin{align*}
E \in R \quad u \in \Sigma^* & \quad v \in \parallel E \xrightleftharpoons{s} u \\
(\text{const } E \ u \ v).get c &= u \\
(\text{const } E \ u \ v).put a s d &= s, d \\
(\text{const } E \ u \ v).create a d &= v, d \\
(\text{const } E \ u \ v).parse c &= c, \{ \}
\end{align*}
\]

The only combinator that is completely new is called match. It takes no arguments, so it is a fully formed S-lens by itself. In the get direction, its only job is to pass off control to the lens chunk. In the put direction, it will have the task of matching up the abstract argument with a corresponding item in the dictionary. It will supply this item to the put function of chunk. We define match along with an auxillary function for retrieving an item from a dictionary and returning a new dictionary that no longer contains the item:

\[
\begin{align*}
\text{lookup}(k, d) &= \begin{cases} 
    c, d[k \leftarrow l] & \text{if } d(k) = c :: l \\
    \text{f}, d & \text{otherwise}
\end{cases}
\end{align*}
\]

Given that, we define match as follows:

\[
\begin{align*}
\text{match} \in \{ \parallel \} & \xrightleftharpoons{s} R_A \\
\text{match.get } c &= \text{chunk.get } c \\
\text{match.put } a s d &= (\text{chunk.put } a c) \cdot d' \\
\text{where } c, d' &= \text{lookup}(\text{chunk.akey } a, d) \\
\text{match.create } a d &= \text{match.put } a \parallel d \\
\text{match.parse } c &= \parallel, \{(\text{chunk.ckey } c) \mapsto c\}
\end{align*}
\]

We present a proof that this combinator is truly a S-lens at the type described by its typing rule, after stating a key lemma about the lookup(\cdot, \cdot) function.

6.1 Lemma: Let \( l \in C \xrightleftharpoons{s} A, c \in C, \) and \( d \in D \). Define \( k = l.ckey c \) and \( d_c = \{(l.ckey c) \mapsto [c]\} \). Then

\[
\text{lookup}(l.ckey (l.get c), d_c, d) = c, d
\]

6.2 Lemma: match \in \{ \parallel \} \xrightleftharpoons{s} R_A.

Proof: SGETPUT: Let \( c \in R \) and \( d, d' \in D \) with \( \parallel, d = \text{match.parse } c \). Then \( d = \{(\text{chunk.ckey } c) \mapsto c\} \) by GETKEY for chunk. Using this fact, with Lemma 6.1 we have

\[
\text{lookup}(\text{akey } (\text{chunk.get } c), d, d'') = c, d'.
\]

We then calculate

\[
\begin{align*}
\text{match.put } (\text{match.get } c) \parallel (d \parallel d') &= \text{match.put } (\text{chunk.get } c) \parallel (d \parallel d') \\
&= (\text{chunk.put } (\text{chunk.get } c), d', d'') \\
&\text{using the fact above; } \\
&= c, d' \quad \text{by SGETPUT for chunk.}
\end{align*}
\]

SPUTGET: Let \( a \in R_A \) and \( c' \in R \) and \( d, d' \in D \) with \( c', d' = \text{match.put } a \parallel d \). Then there exists \( c \in R \) with \( c, d' = \text{lookup}(\text{chunk.akey } a, d) \) and \( c' = \text{chunk.put } a c \). We then calculate

\[
\begin{align*}
\text{match.get } c' &= \text{chunk.get } c' \\
&= a \quad \text{by SPUTGET for chunk.}
\end{align*}
\]

SCREATEGET: Let \( a \in R_A \) and \( c' \in R \) and \( d, d' \in D \) with \( c', d' = \text{match.create } a d \). Then \( c', d' = \text{match.put } a \parallel d \) and there exists \( c \in R \) such that \( c, d' = \text{lookup}(\text{chunk.akey } a, d) \) and \( \text{chunk.put } a c = c' \). As above, we calculate

\[
\begin{align*}
\text{match.get } c' &= \text{chunk.get } c' \\
&= a \quad \text{by SCREATEGET for chunk.}
\end{align*}
\]

The combinator for concatenation takes two lenses as before and has a similar typing rule—in this case we must ensure that both the skeleton languages and their completions can be unambiguously concatenated. The interesting thing to notice about this definition is that it must take the dictionary passed into the put function and thread it through the put function of the sublenses in sequence. The SGETPUT essentially forces this behavior.
Moreover, since $l_1.l_2$, we know that $l_1 \cdot a_i = a_i$ for $i \in \{1, 2\}$. By the definition of $(l_1 \cdot l_2).put$, we know that $(l_1 \cdot l_2).put a s d_0 = (c_1 \cdot c_2), d_2$. Then, we calculate as follows:

$(l_1 \cdot l_2).get (c_1, c_2) = (l_1, get c_1) \cdot (l_2, get c_2) = a_1, a_2 = a$

**SCREATEGet**: Let $a = a_1 \cdot a_2 \in A_1 \cdot A_2$, with $a_1 \in A_1$ and $a_2 \in A_2$. Let $d_0 \in D$. Let $c_1, d_i = \text{l.put} a_i s_i d_{i-1}$ for $i \in \{1, 2\}$. By the CREATEGet law for $l_1$ and $l_2$, we know that $l_1, get c_1 = a_i$ for $i \in \{1, 2\}$. By the definition of $(l_1 \cdot l_2).put$, we know that $(l_1 \cdot l_2).put a s d_0 = (c_1 \cdot c_2), d_2$. Then, we calculate as follows:

$(l_1 \cdot l_2).get (c_1, c_2) = (l_1, get c_1) \cdot (l_2, get c_2) = a_1, a_2 = a$
7. Example

We now develop a longer example of programming with resourceful lenses—a lens mapping between XML inputs representing business cards and simplified ASCII views of the same data. (This example is borrowed from XSugar Brabrand et al. (2005), where the primary goal is providing syntax for bijective transformations between XML and ASCII formats—up to whitespace, which is assumed to be inessential.) We have implemented a prototype interpreter for the combinators described in the previous sections; each of input/output pairs displayed in teletype font this paper have been generated using the implementation. The implementation is based on a library for regular expressions that does not provide a routine for deciding non-ambiguity; thus, we have not yet implemented a static checker.

We begin by defining a lens that only discards XML tags and whitespace, then show how it can be extended to elide parts of the data. The concrete XML documents are sequences of business cards, each represented in the following format:

```
<card>
  <name>Sergey Brin</name>
  <title>Co-Founder Google</title>
  <email>sergey@google.com</email>
  <phone>(415) 555-1111</phone>
  <web url="http://www.google.com"/>
</card>
```

A collection of business cards is just a sequence of individual cards enclosed in `</cards> `. The abstract view of such a sequence is obtained by projecting away white space and rewriting the XML tags to single-characters:

```
"N Sergey Brin
T Co-Founder Google
E sergey@google.com
P (415) 555-1111
W http://www.google.com"
```

To build up this transformation, we first define some regular expressions and lenses describing portions of the input:

```
let URL = /([a-zA-Z0-9.:/~_\ ]+)/
let DATA = /([a-zA-Z0-9()-@.\ ]+)/
let ws = del /\[\n\t\]*/
let cp (t:string) (o:string) (c:string) =
  ws . ("\(<" . t . "\)>") <-> o .
  DATA . ("\(</" . t . ")") <-> c
let k (xml:string) (pre:string) (post:string) =
  ws . ("\(<" . xml . ")") <-> pre.
  key DATA . ("\(<" . xml . ")") <-> post
```

The lens cp is parameterized on the XML tag t and two strings o and c, and maps between \[\n\t\] *t* . d . </t> and o . d . c. (We have not formalized such parameterized lenses above, but they are not problematic: essentially, the language we are actually programming in is a small functional language that has lenses as a base type—and all function applications can be reduced as a first step of execution, leaving a lens. See Foster et al. (2006) for details.) The lens k is identical to cp except that the DATA is wrapped under a key combinator. With these pieces in place, we define a K-lens and an S-lens for processing individual business cards and sequences of cards as follows (to save space, we elide the global default card):

```
let cards_xsugar =
  del "<cards>" .
  (ws | []).
  (ins "\n\n" | [])* . ws).
  del "</cards>"
```

Except for the handling of whitespace, this lens describes the same transformation as the one that can be specified using XSugar. The difference is that, whereas the XSugar transformation from ASCII to XML uses default whitespace, the parser for cards_xsugar actually copies the whitespace (along with the rest of each entry) into the dictionary. Thus, for example, if we put

```
"N Larry Ellison
T CEO Oracle
E larry@oracle.com
P (650) 555-3333
W http://www.oracle.com"
```

into

```
"<card>
  <name>Larry Ellison</name>
  <title>CEO Oracle</title>
  <email>larry@oracle.com</email>
  <phone>(650) 555-2222</phone>
  <web url="http://www.oracle.com"/>
</card>
```

the whitespace for Brin’s entry is copied from the input, while the whitespace for Ellison’s is taken from the global default (which has no indentation). The whitespace for Ellison’s entry is copied from the input, while the whitespace for Ellison’s is taken from the global default (which has no indentation).

```
"<card>
  <name>Larry Ellison</name>
  <title>CEO Oracle</title>
  <email>larry@oracle.com</email>
  <phone>(650) 555-3333</phone>
  <web url="http://www.oracle.com"/>
</card>
```

(If we used just the get and create components of this lens, we would obtain precisely the same transformation as is expressible in XSugar—in going from XML to ASCII and back, whitespace is canonicalized.)

More interesting than simply projecting and restoring whitespace and tags is to define a genuine view that projects away some of the data. We can do this by adding a new (parameterized) lens for deleting a field:

```
let rm (xml:string) =
  del rm xml
```

```
If we update this view by swapping the order of entries and adding in Steve Jobs and Sergey Brin from above, we obtain this abstract view:

"Sergey Brin, Co-Founder Google
Steve Jobs, CEO Apple"

If we update this view by swapping the order of entries and adding in the title

"Steve Jobs, CEO Apple Inc.
Sergey Brin, Co-Founder Google Inc."

and put this view back with the original XML input, we get an updated XML document

"<cards>
  <card>
    <name>Steve Jobs</name>
    <title>CEO Apple Inc.</title>
    <email>steve@apple.com</email>
    <phone>(650) 555-2222</phone>
    <web uri="http://www.apple.com"/>
  </card>
  
  <card>
    <name>Sergey Brin</name>
    <title>Co-Founder Google Inc.</title>
    <email>sergey@google.com</email>
    <phone>(415) 555-1111</phone>
    <web uri="http://www.google.com"/>
  </card>
</cards>"

that accurately reflects the changes in the view.

8. Related Work

Classic lenses form the basis point for this work (Foster et al. 2007). The lens combinators in that work were designed to operate on trees and do not offer any guarantees analogous to those we express through the EQUIVPUT law of our R-lenses. The list-mapping combinator in that work, which aligns elements by position, suffers from the same problems as the lens in the introduction. By specializing our attention to strings, we have been able to design combinators for building lenses that are not restricted to positional alignments during the put operation.

Readers familiar with that work will note some minor technical differences between our presentation of classic lenses and the one given there. We handle the situation where a new element of C must be created from just an element of A using an explicit create function instead of enriching C with a special element Ω that can be used when no real concrete argument is available. Second, since we are not considering lenses defined by recursion, we take the components of lenses to be total functions instead of defining lenses with partial components and establishing totality afterward.

Classic lenses for relational structures have also been developed (Bohannon et al. 2006). A survey of the relationship between classic lenses and the view update problem in the database literature is given in the original lens paper.

Meertens’s formal treatment of constraint maintainers for user interfaces (1998, Section 5.3) recognizes the problem we are dealing with in this paper when operating on lists, and proposes a solution for the special case of “small updates”.

9. Extensions and Future Work

The design we have described makes several simplifying assumptions: the get transformations are all specified using regular operators based on finite state string transductions; S-lenses are parameterized by a single, fixed K-lens; and keys are ordinary strings. In this section, we sketch how these aspects of the design might be generalized.

The get transformations of basic string lens combinators we have described—copy and const closed under the regular operators—are all expressible as simple, one-way finite state transductions. This class contains many useful transformations, powerful enough to express a large collection of examples, but has a fundamental limitation: the restriction to finite state makes it impossible to remember arbitrary amounts of data. For example, if in the business card example the order of the name and title elements were reversed, then we could not transform

"<title>Co-Founder Google</title>
  <name>Sergey Brin</name>"

into:

"Sergey Brin, Co-Founder Google"

Lifting this restriction poses no semantic problems. At the level of syntax, one means of expressing such transformations is to add a combinator swap l₁ l₂ that swaps the views computed by the get functions—i.e., c₁ c₂ maps to (l₂ get c₂)−(l₁ get c₁)—and similarly unswaps the results computed by the put functions. More generally, we could have a version of the iteration combinator that reverses the order of (or, indeed, applies any fixed permutation to) the elements.

Above, we assumed that all S-lenses are defined with respect to a single, globally-fixed K-lens chunk and a fixed default chunk. The match combinator then uses (via dictionaries) a global match-
ing between every chunk in the input and view. In some situations, however, it is useful to define several distinct sorts of chunks, use a different lens to process each sort, and to keep the global matching of different sorts of chunks separate. In our prototype implementation, we generalize the design described above by fixing a set $T$ of string “tags”, providing a tuple of global $K$-lenses (one for each tag), and associating each match combinator to a tag. The type $D$ of dictionaries is similarly generalized from $(K \rightarrow R \text{ list})$ to $(T \rightarrow K \rightarrow R \text{ list})$ where $\text{chunky} \in R_T \iff (R_A)\gamma$—i.e., a dictionary maps tags to keys to lists of data of appropriate concrete type. This allows us to express transformations in which different pieces of the input and view are matched globally, using different dictionaries. This generalization is notionally heavy but unproblematic.

Yet more generally, it would be useful to be able to match data across chunks with different types. For example, suppose that the chunks in the input are either of the form

```
<name>Sergey Brin</name>
<email>sergey@google.com</email>
<title>Co-Founder Google</title>
```

or

```
<name>Sergey Brin</name>
<phone>(415) 555-1111</phone>
```

and that the type of each chunk is determined by the context in which it appears. That is, when it occurs in certain parts of the input, the entry must have an email and phone, while in others, it must have email and title. Because of this constraint, we cannot simply use a single match where chunk is the union of the lenses that process each type of data, because the put function would create entries with email and title in positions where an email and phone are required. Instead, we would like to be able to write lenses where the type of each entry is determined by the context, but the common data—email in this example—still gets matched globally. One way to achieve this behavior is to apply different lenses to the data flowing into and out of dictionaries at occurrences of match; unfortunately, ensuring that the semantic laws are respected seems to impose heavy constraints on such a design. So far, we have only explored some simple extensions in this area—e.g., applying different lenses with get components that are bijections; we leave a full investigation to future work.

Our basic design could also be extended with richer notions of keys—e.g., strings, sets, and lists—as well as combinators for transforming between different kinds of keys. For example, the iteration combinator could return the list of keys of chunks instead of their flat concatenation, and this list could be further transformed into a set, if order is not important, or concatenated into a string. We have experimented with designs incorporating this kind of additional structure, but we do not yet have a feeling for how much is useful in practice.

Another extension that we would like to investigate is the possibility of defining sequential composition for $S$-lenses. The definition of composition given for $R$-lenses in Section 3, together with the construction that converts an $S$-lenses to an $R$-lenses, gives a way to compose two $S$-lenses, but (a) it is rather indirect and (b) it can only be used “at the top-level”—i.e., the result is an $R$-lens, not an $S$-lens. It is not currently clear to us what would be required to compose $S$-lenses without going through $R$-lenses.

Finally, we also plan to develop our prototype implementation to a full-blown string transformation language. We also hope to investigate $R$-lenses for structures besides strings including trees, relations, and graphs.

Acknowledgments

We are grateful to Perdita Stevens, Stijn VanSummeren, Val Tannen, James Cheney, Ravi Chugh, and the members of the UPenn PL Club and Database Group for helpful discussions. Our work has been supported by the National Science Foundation under grant IIS-0534592 Linguistic Foundations for XML View Update. Nathan Foster is also supported by an NSF Graduate Research Fellowship.

References


